Паралельна реалізація алгоритму BFS

Презентацію підготував
Студент групи ІП-11
Панченко Сергій

Об’єкт та предмет дослідження. Призначення та мета розробки

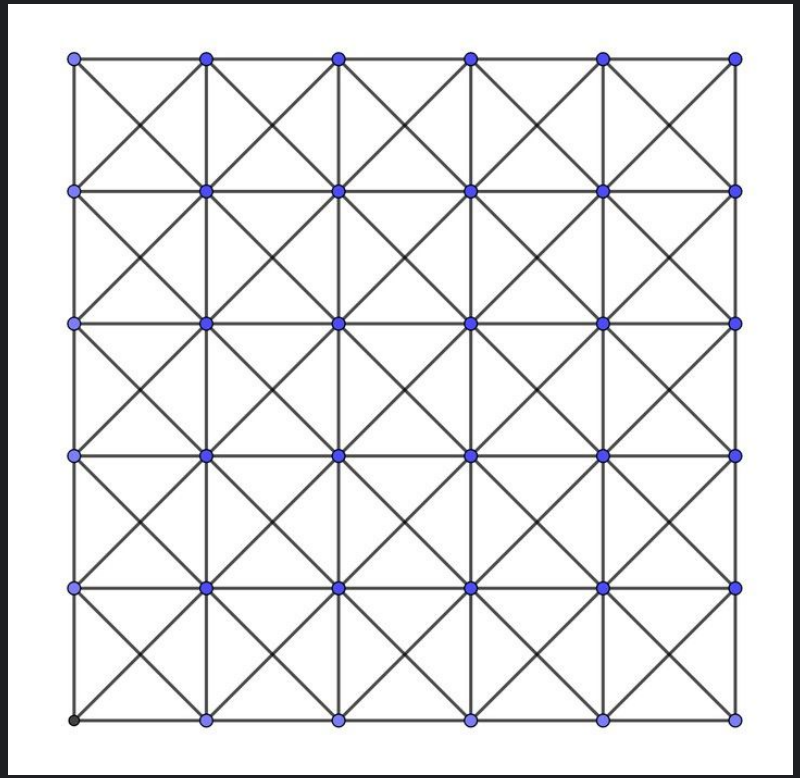
- Об’єктом задача паралельного пошуку шляху у графі.
- Предметом дослідження є паралельна реалізація алгоритму BFS.
- Призначенням є обробка та збереження даних на операційних системах сімейства Unix.
- Метою є теоретично дослідити паралельні методи пошуку шляху у графі за допомогою алгоритму BFS; переглянути відомі їхні реалізації; спроектувати, реалізувати, протестувати послідовний та паралельний алгоритми; дослідити ефективність паралелізації програми

ПОСЛІДОВНИЙ АЛГОРИТМ. ПСЕВДОКОД.

```
Algorithm PredecessorNodesImpl():
    create queue with start node
    visitorMap = CreateVisitorMap()
    isFoundEndNode = false
    while queue is not empty and not isFoundEndNode:
        currentNode = dequeue front node from queue
        for each neighbour in graph.adjacent(currentNode):
        if not visitorMap[neighbour].visited:
            visitorMap[neighbour].visited = true
            visitorMap[neighbour].predecessor = currentNode
            if neighbour == end node:
                isFoundEndNode = true
                break
            queue.push(neighbour)
    if not isFoundEndNode:
    return None
    return visitorMap
```

```
Algorithm CreateVisitorMap():
    visitorMap = new map
    for each vertex in graph:
        visitorMap[vertex] = (visited = false, predecessor = None)
    return visitorMap

Algorithm DeterminePath(predecessorNodes):
    path = [end node]
    currentNode = path's first element
    while currentNode is not start node:
    currentNode = predecessorNodes[currentNode].predecessor
    path.add(currentNode)
    reverse(path)
    return path
```

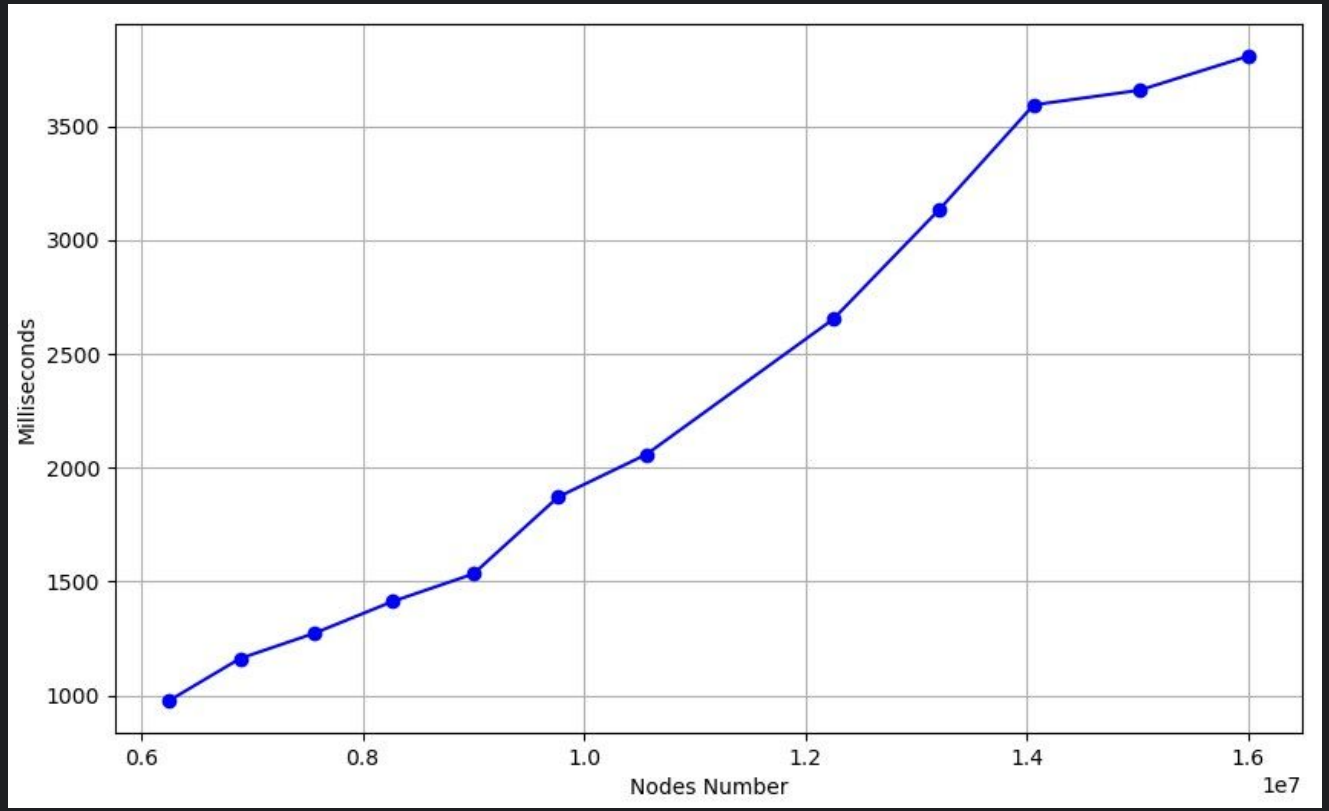
Швидкодія. Вигляд графа та характеристики ПК.



Hardware Model	HP HP 255 G8 Notebook PC
Memory	32.0 GiB
Processor	AMD® Ryzen 5 5500u with radeon graphics × 12
Graphics	RENOIR (renoir, LLVM 15.0.7, DRM 3.49, 6.2.0-33-generic)
Disk Capacity	512.1 GB

OS Name	Ubuntu 22.04.4 LTS
OS Type	64-bit
GNOME Version	42.9
Windowing System	X11
Software Updates	〉

Швидкодія. Результати.



Програмне забезпечення

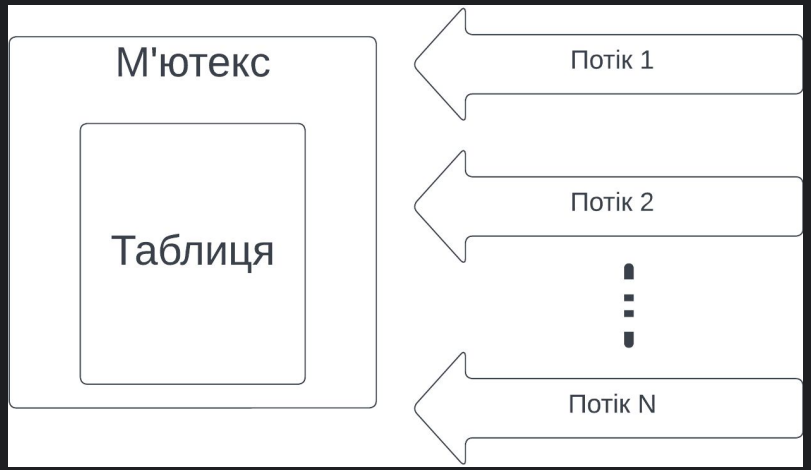
GoogleTest
Тестування програмного забезпечення

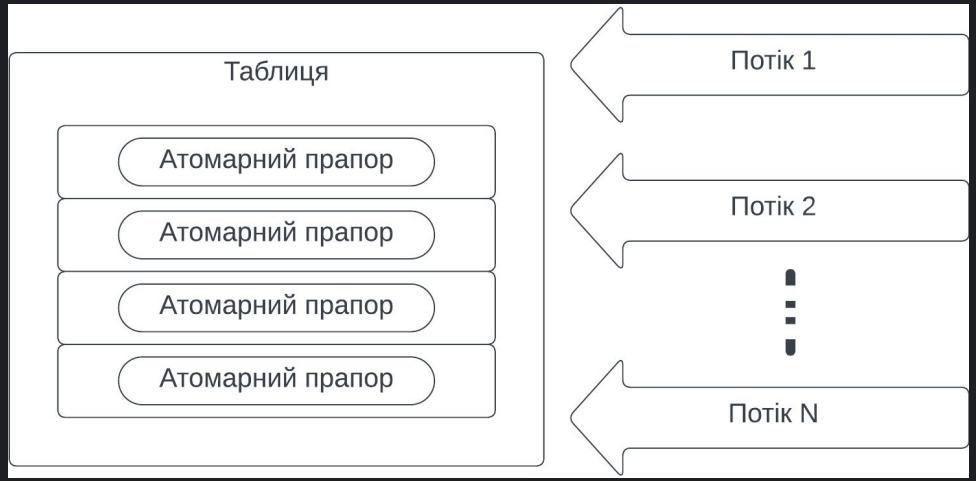
CLion
Середовище розробки C++

CMake
Система збірки C++

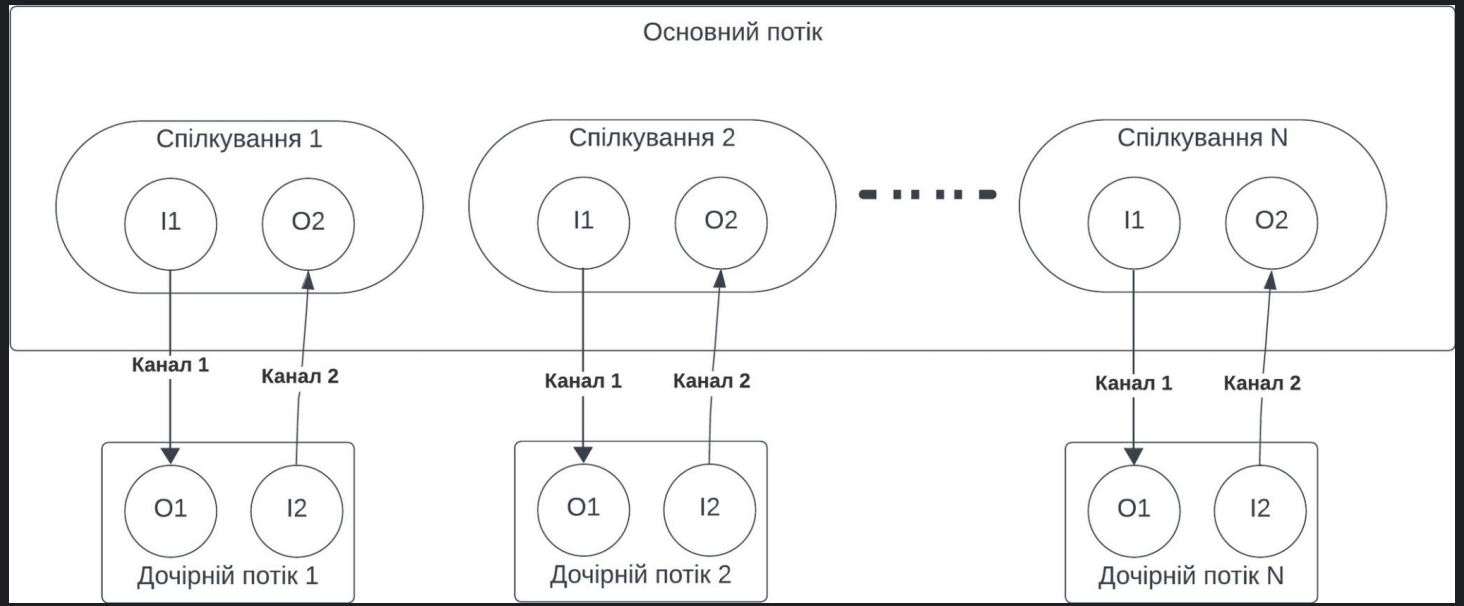
Magic Enum
Зручна робота з enum-типами

Проектування паралельного алгоритму. Таблиця відвідування.





Проектування паралельного алгоритму. Канали.



Реалізація. Псевдокод.

```
FUNCTION CreateVisitorMap
    INITIALIZE visitorMap as empty map
    FOR each node in graph
    INSERT node into visitorMap with initial state
    RETURN visitorMap
```

```
FUNCTION IterateWork(deque, senders, visitorMap)
    INITIALIZE iteration result
    DISTRIBUTE work among child threads and collect results
    UPDATE deque and visitorMap based on children's results
    RETURN iteration result
```

```
FUNCTION Communicate(deque, totalEnqueuedNum, visitorMap, senders, listeners)
    INITIALIZE communication result
    HANDLE messages from parent threads and update the deque and visitorMap
    accordingly
    RETURN communication result
```

```
FUNCTION DoPartialWork(queueView, visitorMap)
    INITIALIZE partial result
    FOR each node in the segment of the deque
        IF node is unvisited
            MARK node as visited and perform necessary actions
    RETURN partial result
```

```
FUNCTION ProcessIterationResult(
    deque, partialResult, senders, totalEnqueued)
    ANALYZE partialResult
    IF end node is found
        RETURN EndNodeFound
    ELSE
        UPDATE deque and continue searching
    RETURN ContinueIteration
```

```
FUNCTION ChildThreadWork(childSender, parentListener, visitorMap)
    WHILE BFS is not complete
        RECEIVE message from parent
        EXECUTE partial BFS work based on the message
        SEND result back to parent
```

Реалізація паралельного алгоритму на C++

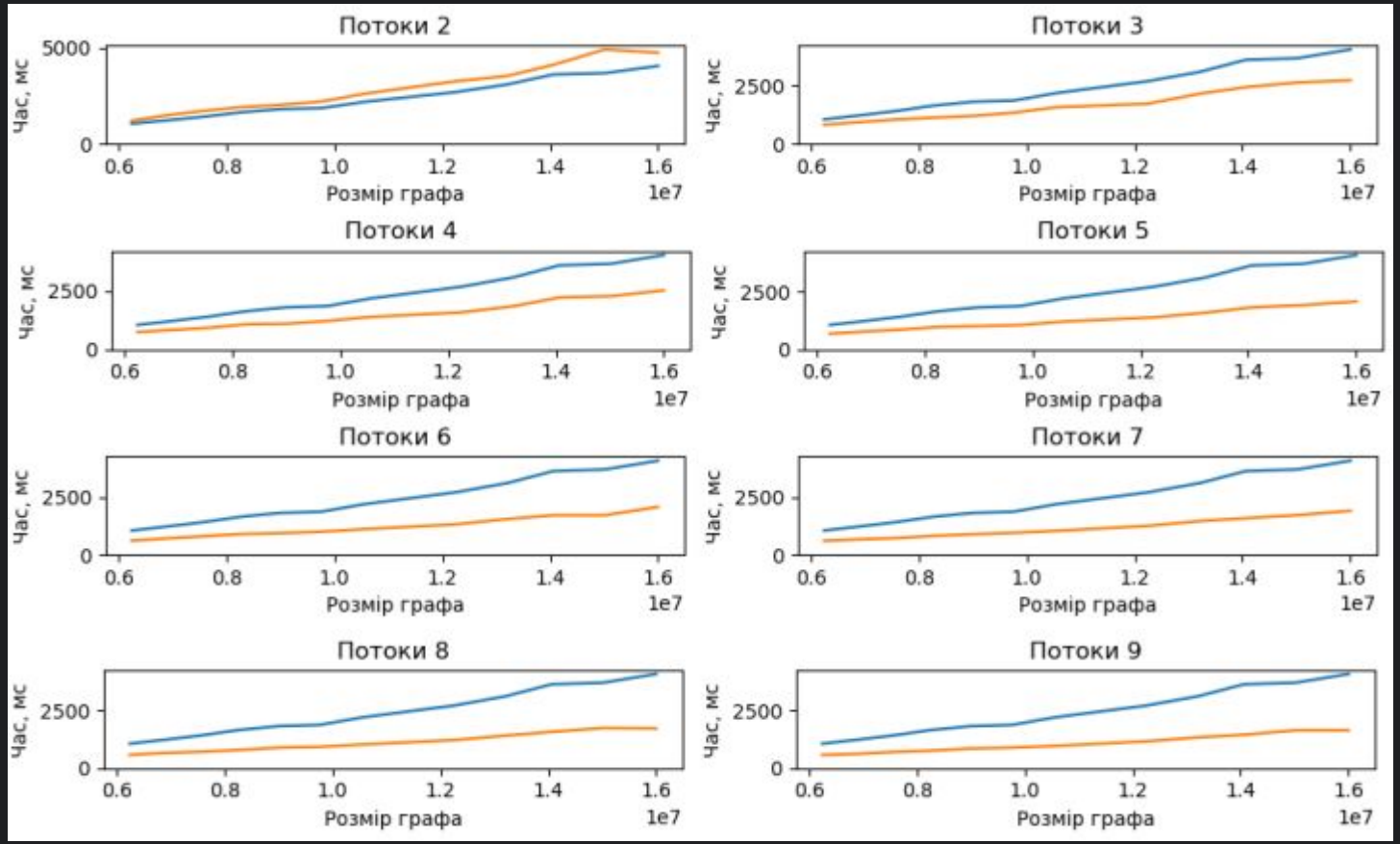
```
struct SContinueIteration {};
struct SEndNodeFound {};
struct SAllNodesEnqueued {};
struct SQueueView {
    const TDeque<T>* Deque;
    size_t Begin;
    size_t End;
};
struct SFrontier {
    std::vector<T> Data;
};
ACommunicationResult Communicate(
    TDeque<T>& deque,
    size_t& totalEnqueuedNum,
    AVisitorMap& visitorMap,
    std::vector<TPipeWriter<AParentMessage>>& senders,
    std::vector<TPipeReader<AChildrenMessage>>& listeners
) const;

AChildrenMessage IterateWork(
    const TDeque<T>& deque,
    const std::vector<TPipeWriter<AParentMessage>>& senders,
    AVisitorMap& visitorMap
) const;
```

```
AChildrenMessage DoPartialWork(
    const SQueueView& queueView,
    AVisitorMap& visitorMap) const;

void ChildThreadWork(
    const TPipeWriter<AChildrenMessage>& childSender,
    const TPipeReader<AParentMessage>& parentListener,
    AVisitorMap& visitorMap
) const;
```

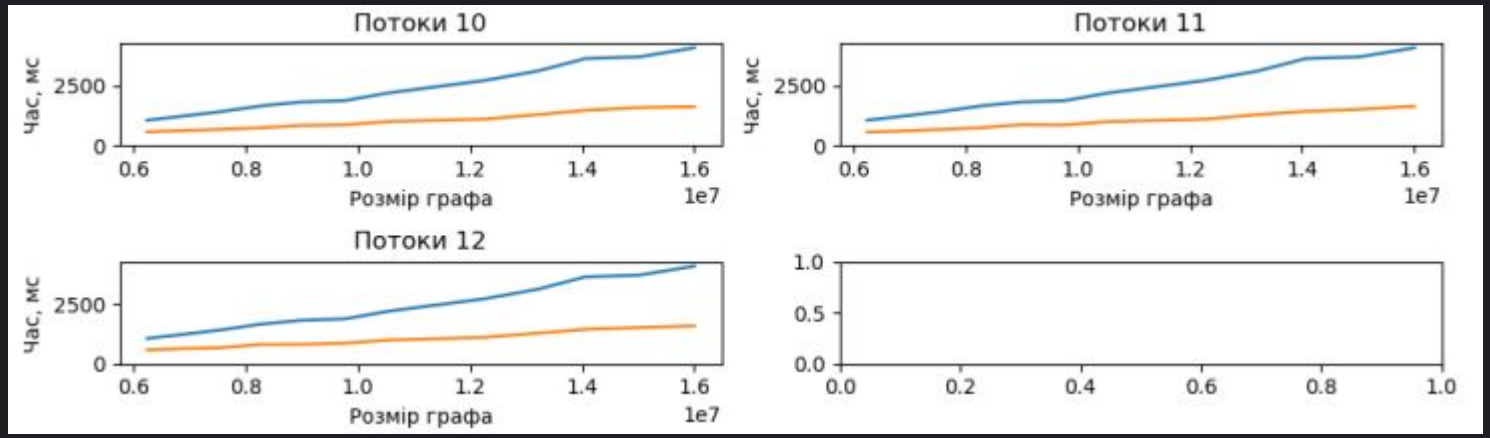
```
using AParentMessage = std::variant<
    SEndNodeFound,
    SAllNodesEnqueued,
    SQueueView>;

using AChildrenMessage = std::variant<
    SEndNodeFound,
    SFrontier>;

using ACommunicationResult = std::variant<
    SAllNodesEnqueued,
    SEndNodeFound
>;

using AIterationResult = std::variant<
    SEndNodeFound,
    SContinueIteration
>;
```

```
auto ProcessIterationResult(
    TDeque<T>& deque,
    AChildrenMessage&& partialResult,
    const std::vector<TPipeWriter<AParentMessage>>& senders,
    size_t& totalEnqueued
) const -> AIterationResult;

template<typename MessageType>
MessageType SendMessageToAll(
    const std::vector<TPipeWriter<AParentMessage>>& senders
) const;
```

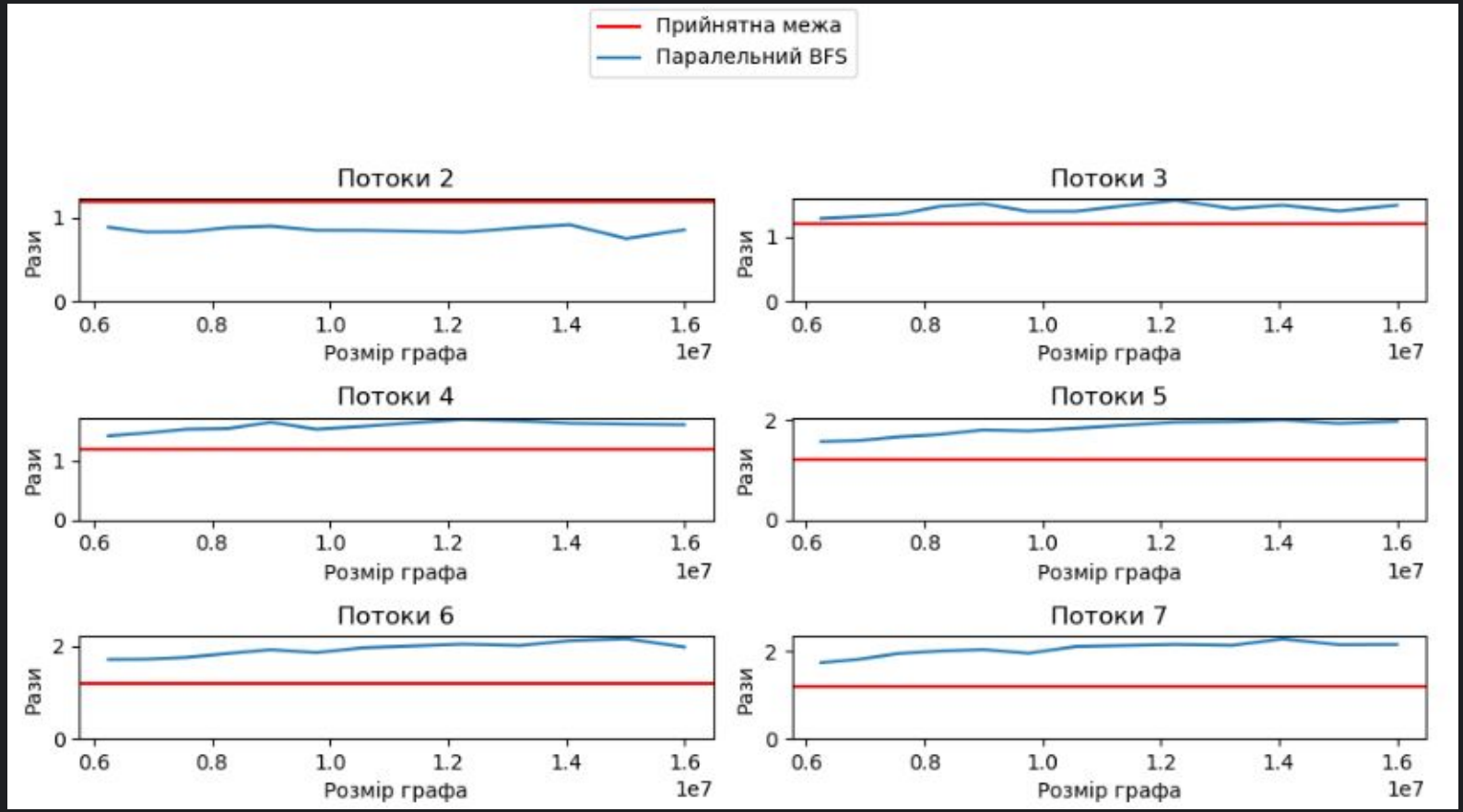
Дослідження ефективності. Залежність часу від розміру.



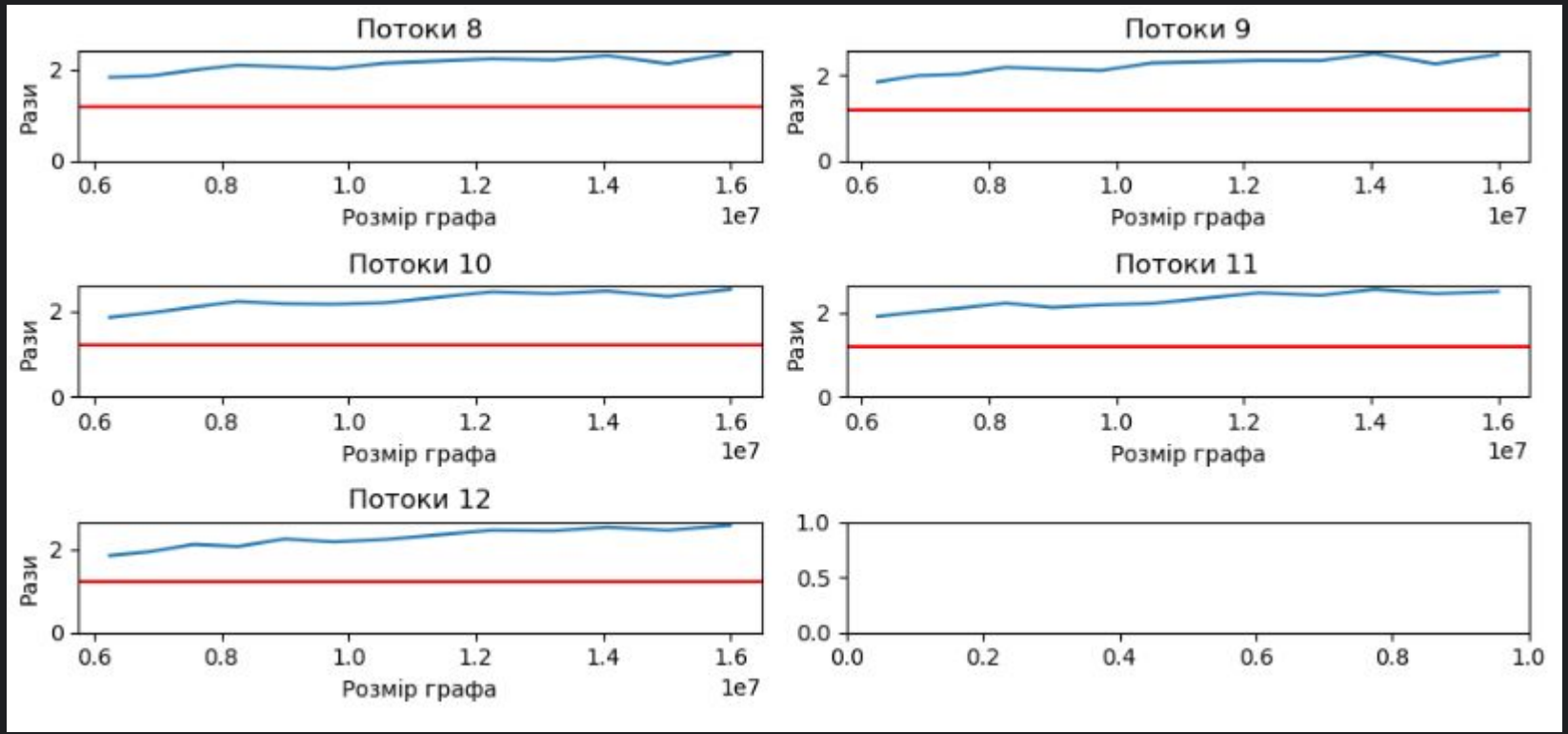
Дослідження ефективності. Залежність часу від розміру.



Дослідження ефективності. Залежність ефективності від розміру.



Дослідження ефективності. Залежність ефективності від розміру.



Аналіз результатів

Паралельний BFS демонструє значне зменшення часу виконання порівняно з послідовним виконанням, особливо для великих графів і при використанні великої кількості потоків. Це підтверджує ефективність паралельної обробки для таких задач.

З зображень видно, що прискорення збільшується з кількістю потоків, але існує межа, після якої додавання додаткових потоків не призводить до значного збільшення прискорення. Це може бути пов'язано з накладними витратами на синхронізацію та управління потоками, які зрештою обмежують загальне прискорення.

Висновок

Аналіз графічних даних, отриманих в ході дослідження, продемонстрував значні переваги застосування паралельного виконання BFS у порівнянні з послідовним. При збільшенні кількості потоків до чотирьох, прискорення склало близько 45%, що свідчить про ефективність паралелізації задачі. При збільшенні кількості потоків до 12 прискорення склало 130%. Оптимальним варіантом виявилося використання восьми потоків, при якому досягнуто прискорення більше ніж у 2 рази порівняно з послідовною реалізацією.

У підсумку, дослідження підтвердило високу ефективність паралельної реалізації BFS, зокрема, при оптимальній кількості потоків, що дозволяє значно скоротити час обробки великих графів.